

Cours & exercices de Python

Bastien ZALUGAS

Contents

Introduction

Manuel de Python

Tu peux télécharger le manuel de python en cliquant [ici](#).

Présentation

- Prénom
- As-tu déjà programmé ? Si oui, avec quel langage de programmation ?
- Motivation : pourquoi t'es-tu inscrit à ce cours ?

Notions de base

- Programmation ?
- Programme ?
- Langage de programmation ?
- Instruction ?
- Compilation ?
- Exécution ?
- Syntaxe ?

Pourquoi Python ?

- Langage interprété : versatilité (le fonctionnement ne dépend pas du système d'exploitation)
- Syntaxe simple
- Très utilisé dans pleins de domaines différents

Fonctionnement du cours

Les notions dont tu auras besoin dans ce cours se trouvent dans le **manuel de python**. Dans chaque cours, les chapitres que tu dois lire sont indiqués au début dans la section **Manuel**. Une fois que tu as lu les chapitres demandés, tu devras faire les exercices de la section **Exercices**.

Pour quasiment chaque exercice, on te demande de créer un **nouveau programme**, c'est à dire un **nouveau fichier**. Retiens-bien ça : **Un programme = un fichier**.

Une fois les exercices d'un cours terminés, appelle le professeur pour qu'il vienne vérifier que ce que tu as fait est correct. **N'oublie pas d'essayer ton programme avant pour être sûr que tu as terminé l'exercice et qu'il n'y a pas d'erreur.**

Introduction à IDLE

- Sauvegarder un programme (p.18 du manuel)
- Premier programme : Crée un premier programme nommé `bonjour.py` qui affiche le texte **Bonjour tout le monde !**

Chapitre 1 : Calculs et variables

Cours 1 : Opérateurs arithmétiques

Manuel

- Calculer avec Python, p.22-23
- Les opérateurs de Python, p.23
- L'ordre des opérateurs, p.24-25

Exercices

Pour chaque exercice, il faut créer un nouveau programme.

1. Dans un programme nommé `cours1.py` :
 - Affiche le résultat d'une **addition**.
 - Affiche le résultat d'une **soustraction**.
 - Affiche le résultat d'une **multiplication**.
 - Affiche le résultat d'une **division**.
 - Affiche le résultat d'un calcul utilisant **tous les opérateurs**.
 - Affiche les résultats de **deux calculs contenant les mêmes valeurs et opérateurs** mais avec et sans **parenthèses**, pour illustrer l'ordre des opérateurs et le fonctionnement des parenthèses.

Cours 2 : Variables

Manuel

- Les variables sont comme des étiquettes, p.25-26
- Utiliser les variables, p.26-29

Exercices

1. Dans un programme nommé `variables.py` :
 - Crée une variable nommée `mon_age` égale à ton âge, puis affiche le contenu de cette variable.
 - Crée une variable nommée `annee_actuelle` égale à l'année actuelle.
 - Affiche le résultat de `annee_actuelle` moins `mon_age`.

Chapitre 2 : Chaînes, listes, tuples, dictionnaires

Cours 3 : Chaînes de caractères 1

Manuel

- Les chaînes, p.32
- Créer des chaînes, p.32-33
- Gérer les problèmes de chaînes, p.33-36

Exercices

1. Dans un programme nommé `prenom_nom.py` :
 - Crée une variable `prenom` contenant une chaîne de caractères correspondant à ton prénom.
 - Crée une variable `nom` contenant une chaîne de caractères correspondant à ton nom de famille.
 - Affiche **Bonjour ! Je m'appelle** suivi du contenu de `prenom`, puis d'un espace, puis du contenu de `nom`.
2. Dans un programme nommé `multiligne.py` :
 - Crée une variable `ma_journee` contenant une chaîne de caractères de plusieurs lignes racontant ta journée.
 - Affiche `ma_journee`.
3. Dans un programme nommé `guillemets.py` :
 - Crée une variable `guillemets` qui contient la phrase **C'est vraiment compliqué de dire mon prof de python est le meilleur sans provoquer d'erreur..**
 - Affiche cette variable.

Cours 4 : Chaînes de caractères 2

Manuel

- Insérer des valeurs dans des chaînes, p.36-37
- Multiplier des chaînes, p.37-38

Exercices

1. Dans un programme nommé `prenom_nom_mieux.py` :
 - Crée une variable `prenom` contenant une chaîne de caractères correspondant à ton prénom.
 - Crée une variable `nom` contenant une chaîne de caractères correspondant à ton nom de famille.
 - Affiche **Bonjour ! Je m'appelle** `<prenom>` `<nom>` en remplaçant `<prenom>` et `<nom>` par les variables créées plus tôt et en utilisant `%s`.

2. Dans un programme nommé `mon_salaire.py` :
 - Crée une variable `pieces_par_jour` contenant le nombre de pièces d'or que tu gagnes par jour.
 - Crée une variable `jours_de_travail` contenant le nombre de jours où tu as travaillé.
 - Affiche **En travaillant <jours_de_travail> et avec un salaire quotidien de <pieces_par_jour>, j'ai gagné un total de <jours_de_travail * pieces_par_jour>. Je suis riche !** en utilisant `%s`. Il faudra bien sûr remplacer les parties entre chevrons (`<...>`) par le contenu des variables.
3. Dans un programme nommé `perroquet.py` :
 - Crée une variable `phrase` contenant la phrase de ton choix.
 - Affiche **Le perroquet répète : .**
 - Affiche 10 fois la phrase contenue dans `phrase` sans te répéter.
4. Dans un programme nommé `mur_de_caractères.py` :
 - Crée une variable `caractere` qui contient un caractère de ton choix. Par exemple `#` ou `-` ou ce que tu veux.
 - Crée une variable `repetition` et assigne-lui un nombre entier de ton choix.
 - Affiche autant de fois ton `caractere` que le nombre dans `repetition`.

Cours 5 : Listes 1

Manuel

- Plus puissantes que les chaînes : les listes, p.38-41

Exercices

1. Dans un programme nommé `liste_courses.py` :
 - Crée une variable `courses` contenant une **liste** d'au moins 5 choses à acheter. Chaque élément sera une chaîne de caractères.
 - Affiche la liste de courses.
 - Affiche le 1^{er} élément de la liste en utilisant son indice.
 - Affiche le 3^{ème} élément de la liste en utilisant son indice.

- Modifie le 2^{ème} élément de la liste et l'afficher après modification.
- Modifie le dernier élément de la liste et l'afficher après modification.
- Affiche d'un seul coup le sous-ensemble de la liste contenant les 2^{ème}, 3^{ème} et 4^{ème} éléments de la liste.

2. Dans un programme nommé `liste_courses_quantite.py` :

- Crée une variable `courses_quantite` contenant une **liste** d'au moins 5 chiffres. Après chaque chiffre, place une chaîne de caractères correspondant à quelque chose à acheter.
- Affiche `courses_quantite`.
- Affiche le sous-ensemble contenant les 3^{ème} et 4^{ème} éléments de la liste.

3. Dans un programme nommé `cadavre_exquis.py` :

- Crée une variable `debuts_de_phrases` contenant une **liste** de chaînes de caractères qui pourraient être des débuts de phrase. Par exemple, Aujourd'hui, j'ai décidé de, ou alors Pour la nouvelle année, j'ai pris la décision de .
- Crée une variable `fins_de_phrases` contenant une **liste** de fins de phrases. Par exemple, être sage., devenir riche., ne plus manger de bonbons., etc.
- Crée une **liste** `debuts_et_fins_de_phrases` contenant les deux listes précédemment créées.
- Crée une variable `phrase_entiere` contenant n'importe quel élément de la première liste contenue dans `debuts_et_fins_de_phrases` suivi de n'importe quel élément de la deuxième liste de `debuts_et_fins_de_phrases`.
- Affiche la variable `phrase_entiere` pour voir apparaître une phrase inattendue !

Cours 6 : Listes 2

Manuel

- Ajouter des éléments à une liste, p.41-42
- Supprimer des éléments à une liste, p.42
- Arithmétique de liste, p.42-44

Exercices

1. Dans un programme nommé `liste_eleves.py` :
 - Crée une variable `eleves` contenant la liste des prénoms des personnes présentes à l'atelier Python en omettant le tien.
 - Affiche la liste.
 - Ajoute ton prénom et celui du professeur dans la liste.
 - Affiche la liste modifiée.
 - Supprime le prénom du professeur.
 - Place ton prénom au début de la liste.
 - Affiche la liste finale.
2. Dans un programme nommé `encore_plus_de_listes.py` :
 - Crée une liste `liste_1` avec au moins 3 éléments de ton choix.
 - Crée une liste `liste_2` avec au moins 2 éléments de ton choix.
 - Crée une liste `des_listes` qui est la concaténation des deux autres.
 - Affiche la liste `des_listes`.
 - Fais en sorte que la liste `des_listes` ait 5 fois plus de contenu.
 - Affiche la liste `des_listes`.

Cours 7 : Tuples et dictionnaires

Manuel

- Tuples, p.45
- Dictionnaires, p.45-47

Indication complémentaire sur les dictionnaires

Comme vu dans le manuel, tu peux modifier une valeur dans un dictionnaire simplement en utilisant sa clef :

```
fortunes = {'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000}
print(fortunes)
fortunes['Moi'] = 100
print(fortunes)
```

Résultat :

```
{'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000, 'Ma  
{'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000, 'Ma
```

Si tu veux **ajouter** un élément, c'est-à-dire une clef et une valeur, il faut faire comme si on modifiait la valeur de la nouvelle clef :

```
fortunes = {'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000, 'Ma  
print(fortunes)  
fortunes['Mon chat'] = 100  
print(fortunes)
```

Résultat :

```
{'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000, 'Ma  
{'Elon Musk': 180000000000, 'Jeff Bezos': 114000000000, 'Bill Gates': 104000000000, 'Ma
```

Exercices

1. Dans un programme nommé `mon_tuple.py` :
 - Crée un tuple `un_tuple` avec au moins 3 éléments.
 - Affiche le tuple.
 - Modifie la première valeur du tuple.
 - Affiche le tuple modifié.
2. Dans un programme nommé `courses_dictionnaire.py` :
 - Crée un dictionnaire `courses` contenant 3 éléments à acheter comme clef avec comme valeur la quantité à acheter.
 - Affiche le dictionnaire.
 - Ajoute un élément que tu aurais oublié.
 - Supprime le premier élément du dictionnaire.
 - Modifie la deuxième entrée du dictionnaire.
 - Affiche le dictionnaire.

Cours 8 : Input

Cours

Jusqu'ici, nous avons donné à nos variables des valeurs directement dans le code python. Comment fait-on si l'on souhaite poser une question à l'utilisateur sans qu'il ait besoin de modifier le code ?


La méthode la plus simple, c'est d'utiliser la fonction `input()`. Nous utilisons déjà la fonction `print()` pour afficher du texte à l'écran mais nous verrons plus tard et en détail ce qu'est une fonction en programmation.

En anglais, `input` signifie entrée. La fonction `input()` sert à récupérer une entrée utilisateur, c'est-à-dire les informations que l'utilisateur (celui qui utilise le programme) donne à l'ordinateur. Pour le moment, les informations seront seulement des caractères que l'utilisateur tapera au clavier. Lorsque nous utiliserons `input()`, l'utilisateur pourra entrer des caractères au clavier et terminer par la touche <Enter> ou <Entrée> pour terminer. Dans le programme, nous pourrons récupérer ce que l'utilisateur a entré dans une variable pour le réutiliser.

On peut utiliser la fonction `input()` en laissant les parenthèses vides. On peut aussi y placer un message explicatif destiné à l'utilisateur. Par exemple :

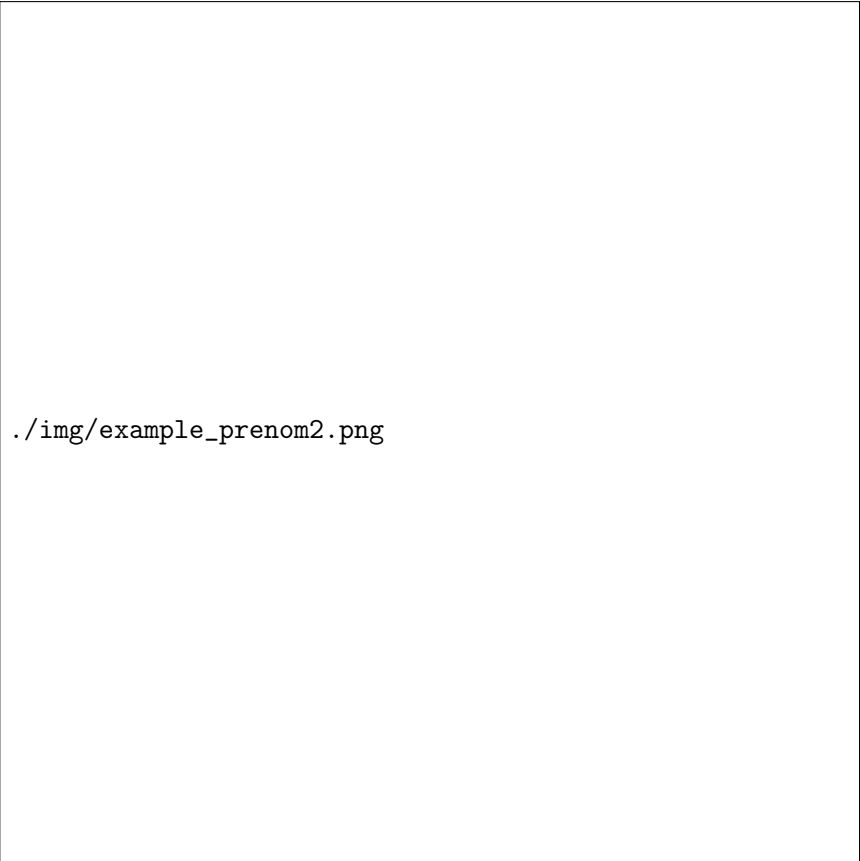
```
prenom = input("Entrez votre prénom : ")
print("Bonjour %s !" % prenom)
```

Résultat :



```
./img/example_prenom1.png
```

L'utilisateur peut entrer son prénom et appuyer sur la touche
<Entrée> :



```
./img/example_prenom2.png
```

La variable dans laquelle nous récupérons l'entrée utilisateur sera une chaîne de caractères. Si nous essayons de faire des opérations arithmétiques avec la valeur récupérée, cela va générer une erreur `TypeError`:

```
age = input("Entrez votre âge : ")  
print(age + 10)
```

Résultat :

```
TypeError: can only concatenate str (not "int") to str
```

Ou encore :

```
age = input("Entrez votre âge : ")  
print("Tu es né en %s !" % (2022 - age))
```

Résultat :

`TypeError: unsupported operand type(s) for -: 'int' and 'str'`

Python génère une erreur car la variable `age` est une chaîne de caractères et non pas un nombre avec lequel on peut faire une opération. Pour résoudre ce problème, il nous suffit de convertir la valeur de retour de `input()` en nombre. Si nous voulons un nombre entier (sans virgule), il faut utiliser `int()`. Si nous souhaitons récupérer un nombre décimal (à virgule), nous devons utiliser `float()`. Exemple pour convertir en un nombre entier :

```
age = int(input("Entrez votre âge : "))
print("Tu es né en %s !" % (2022 - age))
```

Pour le moment, cela suffira pour créer des programmes plus interactifs, mais nous reverrons plus en détail les conversions en python dans le chapitre 4 (cours 13).

Exercices

1. Dans un programme nommé `carte_identite.py` :

- Pour chaque information à demander à l'utilisateur, il faudra utiliser `input()` avec une question appropriée.
- Récupère dans une variable `prenom` le prénom de l'utilisateur.
- Récupère dans une variable `nom` le nom de l'utilisateur.
- Récupère dans une variable `age` l'âge de l'utilisateur.
- Récupère dans une variable `taille` la taille de l'utilisateur.
- Récupère dans une variable `nationalite` la nationalité de l'utilisateur.
- Affiche **Je m'appelle <prenom> <nom>, j'ai <age> ans, je fais <taille>cm et mon pays d'origine est <nationalite>.**

Chapitre 3 : Turtle

Cours 9 : Turtle

Manuel

- Dessiner avec une tortue, p.49-56

Exercices

1. Dans un programme nommé `carré.py`, fais dessiner un carré à une tortue.
2. Dans un programme nommé `hexagone.py`, fais dessiner un hexagone à une tortue.
3. Dans un programme nommé `etoile.py`, fais dessiner une étoile à une tortue.
4. Dans un programme nommé `etoile_variable.py`, fais dessiner une étoile à 5 branches à une tortue. Cependant, cette fois-ci il doit y avoir une variable nommée `taille_cote` qui doit être utilisée pour déterminer la taille des côtés de l'étoile. Par exemple, si `taille_cote` vaut 1, l'étoile sera composée de segments de 1 pixel de long. Si `taille_cote` vaut 100, l'étoile sera composée de segments de 100 pixels.
5. Dans un programme nommé `visage.py`, fais dessiner un visage à une tortue. Le visage doit contenir au moins deux yeux, un nez et une bouche.

Chapitre 4 : Les conditions

Cours 10 : if

Manuel

- Poser des questions avec `if` et `else`, p. 59
- Instructions `if`, p.60
- Un bloc est un groupe d'instructions, p.60-62
- Des conditions pour comparer des choses, p.62-64

Exercices

1. Dans un programme nommé `boutique.py` :
 - Demande à l'utilisateur un nombre entier correspondant à la somme d'argent qu'il a et stocke sa réponse dans une variable `euros`.
 - Si `euros` est supérieure à 2, le programme doit afficher Tu peux t'acheter des chewing-gums puis enlever 2 à `euros`.

- Si **euros** est supérieure à 30, le programme devra aussi afficher Tu peux t'acheter des légos puis enlever 30 à **euros**.
- Si **euros** est supérieure à 350, le programme devra aussi afficher Tu peux t'acheter une playstation puis enlever 350 à **euros**.
- Si **euros** est supérieure à 5000000, le programme devra aussi afficher Tu es trop riche et tu donnes toute ta fortune ! puis **euros** devra être mise à 0.
- Enfin, affiche Il te reste <euros> en remplaçant **euros** par ce qu'il y a dans la variable.

2. Dans un programme nommé **montagnes_russes.py** :

- Demande à l'utilisateur son âge puis sa taille et stocke ses réponses dans des variables **age** et **taille**.
- Crée une condition qui affiche Tu es trop jeune pour faire les montagnes russes. si **age** est strictement inférieure à 7 ans.
- Crée une condition qui affiche Tu as le bon âge pour faire les montagnes russes. si **age** est supérieure ou égale à 7 ans.
- Crée une condition qui affiche Tu es trop petit pour faire les montagnes russes. si **taille** est strictement inférieure à 60cm.
- Crée une condition qui affiche Tu es trop petit pour faire les montagnes russes sans être accompagné. si **taille** est entre 60cm inclus et 80cm exclus.
- Crée une condition qui affiche Tu as la bonne taille pour faire les montagnes russes sans être accompagné. si **taille** est supérieure ou égale à 80cm.

3. Dans un programme nommé **formes.py** :

- Demande à l'utilisateur un nombre entier que tu stockeras dans une variable **nb_cotes**.
- Demande à l'utilisateur un autre nombre entier que tu stockeras dans une variable **taille_cote**.
- Si **nb_cotes** est égale à 3, une tortue doit dessiner un triangle dont les côtés sont égaux à **taille_cote**.
- Si **nb_cotes** est égale à 4, une tortue doit dessiner un carré dont les côtés sont égaux à **taille_cote**.

- Si `nb_cotes` est égale à 5, une tortue doit dessiner un pentagone dont les côtés doivent être égaux à `taille_cote`.
- Si `nb_cotes` est strictement inférieure à 3, affiche Nombre de côtés trop petit..
- Si `nb_cotes` est strictement supérieure à 5, affiche Nombre de côtés trop grand..

Cours 11 : if, elif, else

Manuel

- Instructions `si-alors-sinon`, p.64-65
- Instructions `if et elif`, p.65-66

Exercices

1. Dans un programme nommé `dragon.py` :
 - Affiche Tu te trouves dans une pièce obscure d'un mystérieux château..
 - Affiche Tu dois choisir entre trois portes. Choisis 1, 2 ou 3... et récupère la réponse de l'utilisateur dans une variable `choix`.
 - Si le joueur a choisi la porte 1, affiche Tu as trouvé un trésor, tu es riche !.
 - Si le joueur a choisi la porte 2, affiche La porte s'ouvre et un ogre affamé te donne un coup de massue. Perdu !.
 - Si le joueur a choisi la porte 3, affiche Il y a un dragon dans cette pièce. Le dragon se réveille et te mange. Il te trouve délicieux... Perdu !.
 - Si le joueur a choisi autre chose, affiche Désolé, il faut entrer 1, 2 ou 3..
2. Dans un programme nommé `endroit_secret.py` :
 - Crée une variable `mot_secret` égale au mot de ton choix.
 - Crée une variable `humeur_gardien` égale à un entier entre 0 et 10.
 - Demande à l'utilisateur d'entrer son mot secret et stocke-le dans une variable `mot_utilisateur`.

- En utilisant `if`, `elif` et `else`, implémente les conditions suivantes :
 - Si le `mot_utilisateur` est égal au `mot_secret`, affiche Bienvenue, vous êtes VIP !.
 - Si le `mot_utilisateur` est différent de `mot_secret`, il y a 3 possibilités :
 - * Soit le gardien est de mauvaise humeur (`humeur_gardien` inférieure à 3). Dans ce cas, affiche Veuillez procéder à une vérification des papiers..
 - * Soit le gardien est de bonne humeur (`humeur_gardien` supérieure à 7). Dans ce cas, affiche Bienvenue, mais vous n’êtes pas VIP..
 - * Soit le gardien est neutre. Dans ce cas, affiche Veuillez prendre connaissance des règles avant d’entrer..

Cours 12 : `and`, `or`

Manuel

- Combiner des conditions, p.66

Exercices

1. Dans un programme nommé `montagnes_russes2.py` :
 - Demande à l'utilisateur son âge puis sa taille et stocke ses réponses dans des variables `age` et `taille`.
 - En utilisant seulement deux `if` et en utilisant `and` et `or`, implémente les conditions suivantes :
 - Si `age` est supérieure ou égale à 7 et `taille` est supérieure ou égale à 80, affiche Tu peux faire les montagnes russes..
 - Si `age` est inférieure à 7 ou `taille` est inférieure à 80, affiche Tu ne peux pas faire les montagnes russes..

Cours 13 : Types de données et conversions

Manuel

- Variables sans valeur : `None`, p.66-67
- Différence entre chaînes et nombres, p.67-69

Exercices

1. Dans un programme nommé `types_donnees.py` :
 - Demande à l'utilisateur un nombre entier et stocke-le dans une variable `entier_str`.
 - Demande à l'utilisateur un nombre décimal et stocke-le dans une variable `decimal_str`.
 - Demande à l'utilisateur un mot de son choix et stocke-le dans une variable `mot`.
 - Convertis `entier_str` en entier dans une variable `entier`.
 - Convertis `decimal_str` en nombre décimal dans une variable `decimal`.
 - Si le nombre entier est égal à ton âge, affiche Bravo tu es tombé sur mon âge !.
 - Si le nombre décimal est égal à 23.23, affiche Bravo tu as trouvé le nombre auquel je pensais !.
 - Si le mot est égal à ton prénom, affiche Bravo tu as trouvé mon prénom !.
 - Demande à nouveau un nombre entier à l'utilisateur et stocke-le directement sous forme de nombre entier dans une variable `entier2`.
 - Affiche le contenu de `entier2`.
 - Essaie ton programme en entrant un mot au lieu d'un nombre entier au début. Que se passe-t-il ?

Chapitre 5 : Les boucles

Cours 14 : Les boucles for

Manuel

- Tourner en boucle, p.73
- Utiliser les boucles for, p.74-80

Exercices

1. Dans un programme nommé `repetition.py` :

- Demande à l'utilisateur un nombre entier et stocke-le dans une variable `nombre`.
 - Fais une boucle `for` qui affiche `nombre` fois Je n'aurais plus jamais besoin de me répéter.
 - Fais une boucle `for` qui affiche tous les nombres de 0 à `nombre` inclus.
 - Fais une boucle `for` qui affiche tous les nombres de 0 à `nombre` exclus.
 - Crée une liste `liste`.
 - Fais une boucle `for` qui demande `nombre` fois à l'utilisateur d'entrer un mot qui s'ajoutera à la liste.
 - Fais une boucle `for` qui affiche chaque élément de la liste ligne par ligne.
2. Dans un programme nommé `pairs.py` :
 - Affiche tous les nombres pairs de 0 à 20 en utilisant une boucle `for`.
 3. Dans un programme nommé `repetortue.py` :
 - Crée deux variables : `distance` et `angle` et demande à l'utilisateur d'entrer leurs valeurs (nombres entiers uniquement).
 - Dans une boucle `for`, fais avancer la tortue de `distance` pixels et fais-la tourner de `angle` degrés.
 - Essaie différentes valeurs pour `distance` et `angle` pour voir ce qu'il se passe de différent.
 4. Dans un programme nommé `cercle.py` :
 - Fais dessiner à une tortue un cercle à l'aide d'une boucle `for`.
 5. Dans un programme `spirale.py` :
 - Fais dessiner à une tortue une spirale à l'aide d'une boucle `for`.
 6. Dans un programme nommé `fibonacci.py` :
 - Fais un programme qui calcule les 10 premiers nombres de la suite de fibonacci en utilisant une boucle `for`. La suite de Fibonacci est : 1, 1, 2, 3, 5, 8, ... Les deux premiers nombres de la suite sont toujours 1 et 1, puis ensuite chaque nombre est égal aux deux précédents.

- (BONUS) : Fais avancer une tortue de chaque valeur de la suite de Fibonacci en la faisant tourner à chaque fois de 90°.

Cours 15 : Boucles while

Manuel

- Tant que nous parlons de boucles : **while**, p.81-83

Exercices

1. Dans un programme nommé `repetition_while.py` :
 - Demande à l'utilisateur un nombre entier et stocke-le dans une variable `nombre`.
 - Fais une boucle **while** qui affiche `nombre` fois Je n'aurais plus jamais besoin de me répéter.
 - Fais une boucle **while** qui affiche tous les nombres de 0 à `nombre` inclus.
 - Fais une boucle **while** qui affiche tous les nombres de 0 à `nombre` exclus.
 - Crée une liste `liste`.
 - Fais une boucle **while** qui demande `nombre` fois à l'utilisateur d'entrer un mot qui s'ajoutera à la liste.
 - Fais une boucle **while** qui affiche chaque élément de la liste ligne par ligne.
2. Dans un programme nommé `compte_a_rebours.py` :
 - Demande à l'utilisateur un nombre entier de son choix et stocke-le dans une variable `nombre`
 - Fais une boucle **while** qui, tant que `nombre` est strictement supérieur à 0 :
 - Affiche `nombre - 1`
 - Demande à l'utilisateur s'il veut continuer. Si l'utilisateur tape '**non**', alors le programme s'arrête.
 - Sinon, `nombre` prend la valeur `nombre - 1`

Chapitre 6 : Mise en pratique

Cours 16 : Nombre secret

Indication pour générer un nombre aléatoire

Pour générer un nombre aléatoire, tu dois d'abord importer le module `random`. Il faut toujours importer les modules en début de programme pour plus de clarté :

```
import random
```

Ensuite, pour générer le nombre et le stocker, tu dois utiliser la fonction `random.randint(min, max)`. `min` et `max` sont les valeurs minimale et maximale du nombre aléatoire souhaité.

```
nombre = random.randint(0, 10)
```

Exercice

Dans un programme nommé `nombre_secret.py` :

- Fais un jeu où l'ordinateur choisit un nombre aléatoire et le joueur doit le deviner.
- Le programme doit choisir un nombre aléatoire entre 1 et 10 en utilisant la fonction vue avant.
- Le joueur doit avoir 4 essais pour deviner le nombre.
- Une fois qu'une partie est terminée (gagnée ou perdue), le programme doit demander à l'utilisateur s'il souhaite rejouer et relancer une partie si celui-ci répond "oui".
- Le nombre d'essai et le minimum et maximum du nombre secret doivent être stockés dans des variables au début du programme et qui doivent être utilisées au bon endroit dans le programme afin de permettre de choisir le nombre d'essai, le minimum et le maximum en changeant seulement la variable au début du programme.
- A chaque essai, le programme doit :
 - Dire au joueur entre combien et le minimum et maximum du nombre secret.
 - Demander au joueur un nombre.

- Si le nombre entré par le joueur est le bon, le jeu doit féliciter le joueur.
- Si le nombre entré est plus petit ou plus grand, le jeu doit informer le joueur et commencer un nouvel essai s’il en reste au joueur.
- Si le nombre n’est pas le bon est qu’il s’agissait du dernier essai, le jeu doit informer le joueur qu’il a perdu.

BONUS

- Si le joueur a gagné, le jeu doit se relancer avec une plus grande difficulté (moins d’essais, nombre secret maximum plus grand, ...)
- Si le joueur a perdu, le jeu doit baisser la difficulté ou retourner à la difficulté originale.
- Des messages cachés qui ne s’affichent que si l’on réussit un certain niveau ou si on trouve le nombre secret du premier coup.

Chapitre 7 : Les fonctions

Cours 17 : Fonctions

Manuel

- **Recycler du code avec des fonctions et des modules**, p.87-88
- **Utiliser des fonctions**, p.88-89
- **Qu’est-ce qu’une fonction ?**, p.89-90
- **Variables et portée**, p.90-92

Exercices

1. Dans un programme nommé `fonctions1.py` :
 - Définir une fonction ayant la signature `bonjour(prenom, nom)` qui doit afficher `Bonjour <prenom> <nom> !`, en remplaçant `<prenom>` et `<nom>` par les arguments passés à la fonction.
 - Définir une fonction `somme(nombre1, nombre2)` qui retourne la somme des deux arguments.

- Définir une fonction `minimum(nombre1, nombre2)` qui retourne le plus petit nombre entre les deux arguments.
- Définir une fonction `puissance(nombre1, nombre2)` qui retourne `nombre1` puissance `nombre2`.
- Définir une fonction `fibonacci(n)` qui retourne le n-ième nombre de la suite de Fibonacci. Exemples : `fibonacci(0) -> 1`, `fibonacci(1) -> 1`, `fibonacci(2) -> 2`, `fibonacci(3) -> 5`, etc.
- Définir et appeler une fonction `menu()` qui :
 - Affiche le nom de toutes les fonctions précédemment créées et demande à l'utilisateur quelle fonction il veut utiliser.
 - Si l'utilisateur entre 'bonjour' :
 - * Demande à l'utilisateur son prénom et son nom, stocke-les et appelle la fonction `bonjour(prenom, nom)` avec ce qu'a entré l'utilisateur
 - Si l'utilisateur entre 'somme', 'minimum' ou 'puissance' :
 - * Demande à l'utilisateur deux nombres et stocke-les dans des variables.
 - * Appelle la fonction `somme(nombre1, nombre2)`, `minimum(nombre1, nombre2)` ou `puissance(nombre1, nombre2)` en fonction du choix de l'utilisateur.
 - Si l'utilisateur entre 'fibonacci' :
 - * Demande à l'utilisateur un nombre et appelle la fonction `fibonnaci(n)` avec celui-ci
 - Tant que l'utilisateur n'entre pas 'quitter', il faut que la fonction propose les choix à nouveau à l'utilisateur.

2. Dans un programme nommé `forme.py` :

- Définis une fonction `forme(nb_cotes, taille_cotes)` qui dessine une forme en utilisant une tortue. La forme doit avoir `nb_cotes` en nombre de côtés et la taille doit être égale à `taille_cotes`.
- Appelle plusieurs fois la fonction avec des arguments différents pour tout bien tester.

3. Dans un programme nommé `forme_aleatoire.py` :

- Définis une fonction ayant la signature `forme_aleatoire(tortue)`.

- La fonction doit définir une variable **repetition** égale à un nombre aléatoire entre 50 et 200.
- La fonction doit définir une variable **distance** égale à un nombre aléatoire entre 10 et 200.
- Le programme doit définir une variable **angle** égale à un nombre aléatoire en 0 et 360.
- La forme aléatoire doit être dessinée par la **tortue** passée en argument en répétant **repetition** fois le fait d'avancer de **distance** pixels et de tourner d'**angle** degrés.
- Pour tester, appelle plusieurs fois cette fonction avec des tortues différentes.

4. Dans un programme nommé **inverse.py** :

- Définis une fonction ayant la signature **inverse(liste)** qui doit renvoyer la **liste** passée en argument à l'envers. Par exemple, si on lui donne la liste [1, 5, 2, 6] en argument, elle doit renvoyer [6, 2, 5, 1].
- Appelle la fonction plusieurs fois avec des listes différentes pour bien tester tous les cas.
- Essaie de passer une chaîne de caractères en argument. Que se passe-t-il ?

5. Dans un programme nommé **minimum_liste.py** :

- Définir une fonction ayant la signature **minimum_liste(liste_nombres)** qui prend une liste de nombre en paramètre et renvoie le plus petit nombre de la liste.
- Appeler plusieurs fois la fonction avec différentes listes de nombres pour bien tester tous les cas.

6. Dans un programme nommé **tri.py** :

- Faire une fonction ayant la signature **tri(liste_nombres)**.
- Le paramètre **liste_nombres** sera une liste de nombres.
- La fonction doit renvoyer une liste qui contient tous les nombres de **liste_nombres** triés dans l'ordre croissant.
- Appeler plusieurs fois la fonction avec différentes listes de nombres pour tester tous les cas.

(BONUS)

- Définis une fonction `creerListe(taille)` qui crée une liste de taille `taille` en demandant à l'utilisateur chaque nombre de la liste et renvoie la liste ainsi créée.
- Appelle la fonction `creerListe` après avoir demandé la `taille` à l'utilisateur puis utilise la liste créée dans la fonction `tri`. Affiche la liste avant et après avoir utilisé `tri`.

Cours 18 : Les modules

Cours

Un **module** sert à regrouper des fonctions, des variables et d'autres choses dans des programmes plus vastes et plus puissants. Certains modules sont intégrés dans Python lui-même, tandis que tu peux en télécharger d'autres de manière séparée. Nous en avons déjà utilisé par exemple avec **turtle** et **random**.

Pour utiliser un module, il suffit d'écrire au début du fichier `import nom_du_module`. Dans le cas où tu voudrais utiliser un module qui n'est pas intégré à Python, il faudra d'abord le télécharger (ou le coder !).

Maintenant, si je te disais que tu peux créer tes propres modules avec tes propres fonctions, variables, classes dedans !? Ne serait-il pas pratique de pouvoir utiliser sans avoir à la réécrire une fonction qui, par exemple, permet de demander un entier à l'utilisateur et de vérifier qu'il n'y a pas d'erreur ?

Pour créer ton propre module, il suffit de :

1. Créer un nouveau fichier et lui donner le nom de ton module, par exemple `mes_fonctions.py`.
2. Écrire tes fonctions à l'intérieur du fichier, autant que tu veux. **Attention, nous voulons seulement écrire les fonctions, pas les utiliser tout de suite.**
3. Une fois le fichier enregistré, il suffit de le mettre dans le même **dossier** que le programme dans lequel tu vas l'utiliser puis de l'importer avec `import mes_fonctions`.

Exercices

Dans le chapitre suivant, nous allons mettre en pratique les modules et les fonctions ensemble en créant notre propre bibliothèque personnalisée de fonctions !

1. Dans un fichier nommé `mes_fonctions.py` :
 - Crée la fonction `demander_int(message)` qui :
 - demande un entier à l'utilisateur en affichant le message `message`.
 - retourne l'entier convertit en `int`.
 - Crée la fonction `demander_float(message)` qui fonctionne comme la précédente mais avec un `float`.
 - Crée la fonction `demander(message)` qui fonctionne comme les précédentes mais avec une chaîne de caractères.
2. Dans un programme nommé `tests_fonctions.py` :
 - importe ton module `mes_fonctions`
 - teste les 3 fonctions présentes dans ton module

Tu as maintenant créé les 3 premières fonctions de ton propre module, que tu pourras réutiliser chaque fois que tu en as besoin. Tu es libre d'y ajouter les fonctions qui te semblent intéressantes et que tu as déjà écrites, par exemple dans ce chapitre.

Chapitre 8 : Mise en pratique 2

Attention ! Pour chaque exercice, il faudra bien faire la différence entre le code **à l'intérieur des fonctions** et le code qui **utilise les fonctions**.

Avant toute chose, crée un fichier nommé `mes_fonctions.py` qui contiendra les fonctions que l'on peut être amenés à réutiliser plus tard.

Niveau 1

Division euclidienne

Dans le fichier `mes_fonctions.py` :

- Crée une nouvelle fonction `div_euclidienne(a,b)` qui **retourne un tuple** contenant le **quotient** et le **reste** de la division euclidienne de `a` par `b` sous la forme `(quotient, reste)`.

Dans un programme nommé `manips_nombres.py` :

- importe ton module `mes_fonctions`

- **Utilise** la fonction `div_euclidienne()` dans un programme qui demande à l'utilisateur deux nombres entiers et affiche le calcul effectué ainsi que le quotient et le reste de la division euclidienne de ces nombres.

Pair ou impair

Dans le fichier `mes_fonctions.py` :

- Crée une nouvelle fonction `is_pair(nb)` qui **retourne True si nb est pair et False sinon**.

Dans le programme `manips_nombres.py`:

- **Utilise cette fonction** après avoir demandé un nombre entier à l'utilisateur et affiche un texte indiquant si le nombre est pair ou impair.

Diviseurs

Dans le fichier `mes_fonctions.py` :

- Crée une fonction `diviseurs(nb)` qui **retourne la liste** de tous les diviseurs du nombre `nb`.

Dans le programme `manips_nombres.py` :

- **Utilise cette fonction** après avoir demandé un nombre entier à l'utilisateur et affiche tous les diviseurs de ce nombre, séparés par des virgules et sans crochets.

Nombres premiers

Dans le fichier `mes_fonctions.py` :

- Crée une fonction `is_premier(nb)` qui **retourne True si nb est un nombre premier et False sinon**.
- Crée une fonction `nb_premiers(n)` qui **retourne une liste** contenant les `n` premiers nombres premiers.

Dans le programme `manips_nombres.py` :

- **Utilise ces fonctions** après avoir demandé à l'utilisateur le nombre (entier) de nombres premiers à afficher puis affiche-les ; **un par ligne**.

Pour la suite, tu es libre d'ajouter et d'utiliser les fonctions que tu veux à ton module `mes_fonctions` pour ne plus avoir besoin de réécrire à chaque fois les fonctions que tu réutilises.

Cercle

Dans un programme nommé `super_cercle.py` :

- Crée une fonction `surface_cercle(rayon)` qui **retourne un float** correspondant à la surface d'un cercle de rayon `rayon`.
- Crée une fonction `perimetre_cercle(rayon)` qui **retourne un float** correspondant au périmètre d'un cercle de rayon `rayon`.
- Crée une fonction `creer_cercle(rayon)` qui **dessine avec Turtle** un cercle de rayon `rayon`.
- **Utilise toutes ces fonctions** dans un programme qui demande le rayon à l'utilisateur et affiche la surface, le périmètre et le dessin du cercle.

Lettres

Dans un programme nommé `lettres.py` :

- Crée une fonction `afficher_lettres(chaine)` qui **permet d'afficher** chaque lettre d'une chaîne de caractères ; une par ligne.
- **Utilise cette fonction** dans un programme qui demande à l'utilisateur un mot ou une phrase et qui affiche ensuite dans l'ordre chaque lettre une à une.

Exemple d'affichage pour `chaine = "python"` :

```
p
y
t
h
o
n
```

Swap de lettres

Dans un programme nommé `swap_lettres.py` :

- Crée une fonction `swap_lettres(chaine, position1, position2)` qui **échange** les lettres à la position `position1` et `position2` dans la chaîne de caractères `chaine`.

- **Utilise cette fonction** dans un programme qui demande à l'utilisateur une chaîne de caractères puis la position des lettres à échanger. Le programme devra afficher la chaîne avant et après modification.

Niveau 2

Occurrences de lettres

Dans un programme nommé `occurrences.py` :

- Crée une fonction `occurrences(chaine)` qui **renvoie un dictionnaire** contenant comme clefs chaque lettre présente dans la chaîne de caractères `chaine` et comme valeurs le nombre de fois où elle apparaît dans la chaîne.
- **Utilise cette fonction** dans un programme qui demande à l'utilisateur un mot ou une phrase et qui affiche chaque lettre et le nombre de fois où elle apparaît.

Exemple d'affichage pour `chaine = "Bonjour"` :

```
Le caractère " B " apparaît 1 fois.
Le caractère " o " apparaît 2 fois.
Le caractère " n " apparaît 1 fois.
Le caractère " j " apparaît 1 fois.
Le caractère " u " apparaît 1 fois.
Le caractère " r " apparaît 1 fois.
```

Trouver les positions

Dans un programme nommé `positions_lettre.py` :

- Crée une fonction `positions(lettre, chaine)` qui **retourne une liste** contenant les positions de la lettre `lettre` dans la chaîne de caractères `chaine`. Si la lettre n'est pas présente dans la chaîne, la première et seule valeur de la liste devra être -1.
- **Utilise cette fonction** dans un programme qui demande d'abord à l'utilisateur un mot ou une phrase puis la lettre à trouver et affiche toutes les positions où se trouve la lettre dans la chaîne.

Taille des chaînes

Dans un programme nommé `taille_chaines_liste.py` :

- Crée une fonction `taille_chaines(liste)` qui **retourne un dictionnaire** qui associe à chaque chaîne de caractères la valeur de sa taille.
- **Utilise cette fonction** dans un programme qui :
 - Demande à l'utilisateur la taille de la liste à créer.
 - Demande à l'utilisateur d'entrer des mots ou des phrases et les ajoute à une liste.
 - Affiche (à l'aide de la fonction `taille_chaines()`) chaque chaîne de caractères suivi de sa taille.

Nombre de voyelles

Dans un programme nommé `nb_voyelles.py` :

- Crée une fonction `nb_voyelles(chaine)` qui **retourne le nombre de voyelles** présentes dans la chaîne de caractères `chaine`.
- **Utilise cette fonction** dans un programme qui :
 - Demande à l'utilisateur d'entrer une chaîne.
 - Affiche le nombre de voyelles présentes dans la chaîne.

Mot inverse

Dans un programme nommé `mot_inverse.py` :

- Crée une fonction `mot_inverse(mot)` qui **retourne une chaîne** correspondant à l'inverse du mot `mot`. **Attention, le mot original ne doit pas être modifié !**
- **Utilise cette fonction** dans un programme qui :
 - Demande un mot à l'utilisateur.
 - Affiche le mot original et son inverse.

Exemple pour le mot `python` :

```
mot original : python
inverse : nohtyp
```

Palindrome

Un palindrome est un mot dont l'ordre des lettres reste le même qu'on le lise de gauche à droite ou de droite à gauche.

Dans un programme nommé `palindrome.py` :

- Crée une fonction `est_palindrome(mot)` qui **retourne True si le mot est un palindrome et False sinon**.
- **Utilise cette fonction** dans un programme qui :
 - Demande un mot à l'utilisateur.
 - Affiche un texte indiquant si le mot est un palindrome ou non.

Niveau 3

Valeurs communes

Dans un programme nommé `valeurs_communes.py` :

- Crée une fonction `valeurs_communes(liste1, liste2)` qui **retourne une liste** contenant toutes la valeurs communes entre `liste1` et `liste2`.
- **Utilise cette fonction** dans un programme qui :
 - Demande à l'utilisateur des éléments à insérer dans une première liste.
 - Demande à l'utilisateur des éléments à insérer dans une seconde liste.
 - Affiche tous les éléments communs entre les deux listes entrées.

Suppression des doublons

Dans un programme nommé `del_doublons.py` :

- Crée une fonction `del_doublons(liste)` qui **supprime** les éventuels doublons présents dans la liste `liste`.
- **Utilise cette fonction** dans un programme qui :
 - Demande à l'utilisateur des éléments à insérer dans une liste.
 - Affiche la liste.
 - Affiche la liste sans les doublons.

Rendu de monnaie

Nous voulons créer un programme qui permet de calculer un rendu de monnaie et de dire de quels billets et pièces nous avons besoin pour former la somme à rendre. Par exemple, si la somme à payer est de 18.57€ et qu'un client donne 100€, le programme doit pouvoir :

1. Calculer qu'il faut rendre 81.43€
2. Afficher les billets et les pièces à utiliser pour cela. Exemple :

Prix à payer : 18.57€

Rendu : 81.43€

Décomposition :

50€ : 1

20€ : 1

10€ : 1

1€ : 1

0.20€ : 2

0.02€ : 1

0.01€ : 1

Nous considérerons les sommes suivantes pour décomposer les sommes : 500€, 100€, 50€, 20€, 10€, 5€, 2€, 1€, 0.50€, 0.20€, 0.10€, 0.05€, 0.02€, 0.01€. L'algorithme devra bien entendu utiliser le moins de pièces et billets possibles.

Dans un programme nommé `rendu_monnaie.py` :

- Crée une fonction `rendu_monnaie(prix, somme_payee)` qui **retourne la somme à rendre**.
- Crée une fonction `decomposer_somme(somme)` qui **retourne l'ensemble des billets et pièces à utiliser et leur nombre** nécessaires pour décomposer la somme d'argent `somme`. Tu peux par exemple retourner cette information sous forme de dictionnaire.
- **Utilise ces fonctions** dans un programme qui :
 - Demande à l'utilisateur un prix (float)
 - Demande à l'utilisateur la somme qu'il donne (float)
 - Affiche la somme à rendre ainsi que les billets et pièces à utiliser.

Chapitre 9 : Programmation orientée objet

Cours 19 : classes et objets

Manuel

- Classes et objets, p.97-98
- Organiser les choses en classes, p.98-99
- Enfants et parents, p.99-100
- Ajouter des objets aux classes, p.100

Exercices

1. Dans un fichier nommé `college.py` :
 - Crée une classe `Humain` qui pour l'instant ne contiendra que le mot-clef `pass`.
 - Crée une classe `Piece` de la même manière.

Puissance 4



Dans ce chapitre, tu vas mettre en applications tout ce que tu as appris dans les chapitres précédents. Pour cela, tu vas créer un projet de plus grande envergure que les exercices que tu as faits.

Dans cette section, nous allons créer un **Puissance 4**. Nous commencerons par le créer à partir des notions que nous avons vues, donc avec une interface en **ligne de commande**. Par la suite, quand nous étudierons comment créer des interfaces graphiques, nous pourrons revenir sur ce projet pour ajouter des visuels plus intéressants.

Les consignes qui sont données dans ce chapitre sont là pour t'aider à réaliser le projet. Le but est que tu développes le Puissance 4 en t'aidant le moins possible des consignes : dans un projet comme celui-ci, il n'y a pas une seule bonne réponse, chacun peut avoir des idées différentes pour développer et c'est le meilleur moyen de progresser. Si tu es en trop grande difficulté, les étapes décrites dans ce chapitre sont là pour t'aider.

Si tu as des idées d'améliorations ou que tu veux faire quelque chose différemment, tu es libre de les réaliser, c'est ton projet ! Néanmoins, le professeur vérifiera que ton travail est correct et correspond aux bonnes pratiques de Python que nous avons étudiées.

Règles du Puissance 4

Le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille comptant **6 rangées** et **7 colonnes**. Chaque joueur dispose de 21 pions d'une couleur, en général jaune ou rouge. Tour à tour, les deux joueurs placent un pion dans la **colonne** de leur choix, le pion **coulisse alors jusqu'à la position la plus basse possible dans la dite colonne** à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un **alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur**. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

Étape 1 : Définir la structure de notre programme

Dans cette étape, tu ne vas pas encore coder. Nous allons d'abord réfléchir à la structure de notre programme et imaginer son fonctionnement général. Il faut que l'on définisse de quelles fonctions nous aurons besoin pour faire fonctionner notre programme. Parfois, il est utile d'utiliser du pseudo code. Le pseudo code, c'est une façon d'écrire un algorithme de façon presque

naturelle, sans langage de programmation. Par exemple, dans un pseudo code on peu écrire :

```
si (utilisateur écrit "Bonjour") alors
    afficher "Bonjour"
sinon
    afficher "Au revoir"
```

Le pseudo code aide à déterminer quelles actions notre programme devra effectuer.

Notre programme ne comportera que des fonctions. Chaque action devra être effectuée grâce à une fonction. C'est comme ça que l'on programme : ça permet de pouvoir s'y retrouver plus facilement dans le code et de pouvoir mieux repérer nos erreurs.

La seule instruction qui ne sera pas dans une fonction sera celle qui lance le jeu !

Fonctionnement de la grille

La grille du jeu sera représentée par une liste **à deux dimension**, c'est à dire une liste de plusieurs listes. Ce sera donc une liste contenant 6 listes qui contiendront chacune 7 chiffres. Chaque sous-liste de la grille représentera une ligne de la grille. Comme le Puissance 4 contient 7 colonnes, chaque ligne aura donc 7 chiffres. La valeur d'une case de la grille a 3 possibilités :

- **0** : la case est libre.
- **1** : un jeton du joueur 1 est placé ici.
- **2** : un jeton du joueur 2 est placé ici.

Au début, la grille sera remplie de **0** :

```
[[0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0]]
```

Lorsqu'un joueur placera un pion, il devra choisir seulement la colonne. À nous de simuler le fait que le pion tombe jusqu'à la première ligne disponible. Nous aurons une fonction pour cela. Par exemple, si le joueur 1 décide de placer son pion dans la colonne 3 (en partant de 0), la grille contiendra :

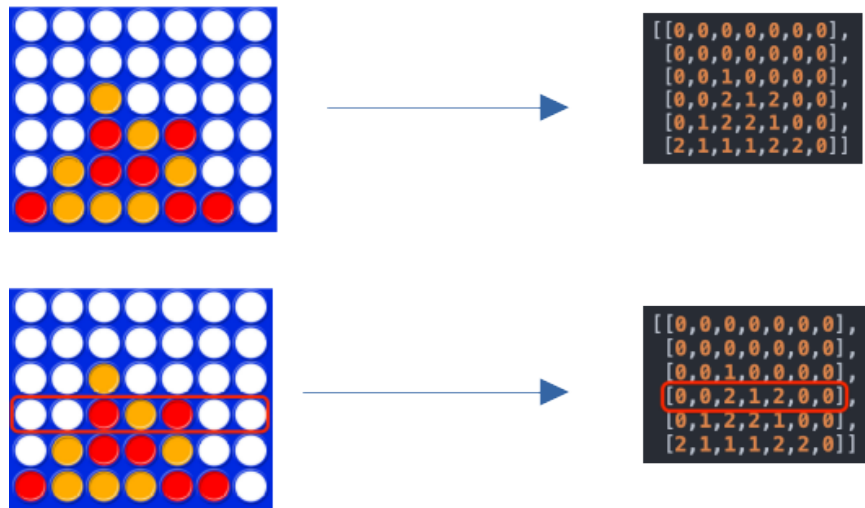
```
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,1,0,0,0]
```

Si le joueur 2 décide de placer son pion dans la même colonne que le joueur 1, la grille deviendra :

```
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0],  
[0,0,0,2,0,0,0],  
[0,0,0,1,0,0,0]
```

Et ainsi de suite.

Par exemple, si les jetons jaunes sont ceux du joueur 1 et les jetons rouges ceux du joueur 2, nous pouvons représenter la grille comme cela :



Pour parcourir une liste à deux dimensions, c'est à dire effectuer un test ou une action sur chaque case de la liste, nous utilisons une double boucle. En effet, par exemple si l'on veut ajouter un pion du joueur 1 dans la 2^{ème} colonne de la 3^{ème} ligne, il faut écrire :

```
grille[2][1] = 1
```

```
# grille[ligne][colonne]
```

Le premier indice correspond à la ligne et le second à la colonne. Alors pour parcourir et afficher toute la liste :

```
for ligne in range(len(grille)):
    for colonne in range(len(grille[ligne])):
        print(grille[ligne][colonne])
```

Liste des fonctions

Maintenant que nous avons une idée générale du fonctionnement de la **grille de jeu**, voici une liste exhaustive des fonctions dont nous aurons besoin. Tu n'as pas besoin de tout comprendre dès le début, cette section servira aussi plus tard pour t'y retrouver dans toutes les fonctions. Ne commences pas non plus à coder les fonctions ici car nous allons revenir en détail sur chacune d'elles dans les étapes suivantes.

L'important est de visualiser à peu près comment sera notre programme.

- **jeu()** : c'est la fonction qui lancera le jeu et le relancera une fois la partie terminée si le joueur en a envie.
 - **initialiserGrille()** : cette fonction permettra d'initialiser (mettre à zéro) la grille du Puissance 4. C'est comme si elle enlevait les pions ! On utilisera cette fonction avant chaque début de partie, dans la fonction **jeu()**.
 - **boucleDeJeu()** : c'est la fonction qui demandera aux joueurs de choisir où poser leurs jetons et les placera tant qu'aucun des deux joueurs n'aura gagné et que la grille n'est pas pleine. Quand le jeu est fini, elle retourne le numéro du joueur qui a gagné (ou **False** si c'est un match nul).
 - * **joueurGagne(joueur)** : cette fonction détermine si le joueur passé en argument a gagné la partie. Elle sera utilisée dans la fonction **boucleDejeu()**.
 - **verifierHorizontalement(joueur)** : vérifie si le joueur passé en paramètre a aligné 4 pions horizontalement dans la grille. Elle sera utilisée dans la fonction **joueurGagne(joueur)**.
 - **verifierVerticalement(joueur)** : vérifie si le joueur passé en paramètre a aligné 4 pions verticalement dans la grille. Elle sera utilisée dans la fonction **joueurGagne(joueur)**.

- **verifierDiagonale(joueur)** : vérifie si le joueur passé en paramètre a aligné 4 pions en diagonale dans la grille. Elle sera utilisée dans la fonction **joueurGagne(joueur)**.
- * **verifierGrillePleine()** : cette fonction devra parcourir la grille pour vérifier s'il y a encore de la place pour poser des jetons. Elle sera utilisée dans la fonction **boucleDeJeu()**.
- * **afficherGrille()** : cette fonction affiche la grille actuelle. Nous formaterons la grille pour qu'elle apparaisse de manière compréhensible aux joueurs. Elle sera utilisée dans la fonction **boucleDeJeu()**
- * **jouerLeTour(joueurActuel)** : cette fonction demandera à un joueur dans quelle colonne il veut placer son jeton. Elle placera ensuite le jeton au bon endroit, c'est-à-dire à la dernière ligne disponible de la colonne demandée. Elle sera utilisée dans la fonction **boucleDeJeu()**
 - **demanderColonne(joueur)** : cette fonction permet de demander au joueur dans quelle colonne il veut placer son pion. Elle renverra le numéro de la colonne demandée. Elle sera utilisée dans la fonction **remplirGrille(joueurActuel)**.
 - **ligneLibreDeLaColonne(colonne)** : cette fonction renverra le numéro de la ligne libre de la colonne choisie par le joueur. Elle sera utilisée dans la fonction **remplirGrille(joueurActuel)**.
 - **placerJeton(joueur, coordonnees)** : cette fonction permet de placer le jeton du joueur passé en paramètre à l'endroit de la grille indiqué par **coordonnees**. **coordonnees** sera un tuple qui aura cette forme : (**ligne, colonne**). Elle sera utilisée dans la fonction **remplirGrille(joueurActuel)**.
- * **recupererSymbole(numeroJoueur)** : cette fonction retourne simplement le symbole correspondant au numéro du joueur passé en paramètre. Si le numéro n'est ni 1 ni 2 alors la fonction renverra un espace ' '. Par exemple, si nous utilisons 'X' pour le joueur 1 et 'O' pour le joueur 2, la fonction **recupererSymbole(1)** renverra 'X'.
- **afficherFinDePartie(gagnant)** : cette fonction fera l'affichage de la fin du jeu en félicitant le gagnant s'il y en a un et en disant qu'il y a match nul sinon.
- **demanderRejouer()** : cette fonction demandera aux joueurs s'ils veulent rejouer et retourne True ou False en fonction de leur réponse.

Étape 2 : Définir les variables globales

Une **variable globale** est une variable qui pourra être utilisée dans toutes les fonctions d'un programme, sans avoir besoin de la passer en paramètre. Pour le Puissance 4, nous aurons besoin de 3 variables. Elles correspondront à **la grille de jeu**, au **symbole du joueur 1** et au **symbole du joueur 2** dans la grille.

Comme vu précédemment, **La grille** sera une liste de 6 listes qui contiendront chacune 7 chiffres. Chaque sous-liste de la grille représentera une ligne de la grille.

Pour les symboles, je te conseille d'utiliser les caractères '**X**' et '**O**' pour bien différencier les joueurs.

Pour créer des variables globales, c'est très simple : c'est comme des variables normales, sauf qu'elles ne sont pas dans une fonction. Pour notre cas, nous appellerons nos variables `grille`, `symboleJoueur1` et `symboleJoueur2` :

```
grille = [[0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0],
           [0,0,0,0,0,0,0]]
symboleJoueur1 = 'X'
symboleJoueur2 = 'O'
```

Nous n'aurons pas besoin d'autres variables globales pour le moment.

Étape 3 : Définir les fonctions d'affichage

Affichage des symboles des joueurs

- Crée une fonction `recupererSymbole(numeroJoueur)`. La logique de cette fonction sera :

```
si numeroJoueur est égal à 1:
    renvoyer symboleJoueur1
sinon si numeroJoueur est égal à 2:
    renvoyer symboleJoueur2
sinon
    renvoyer le caractère 'espace'
```

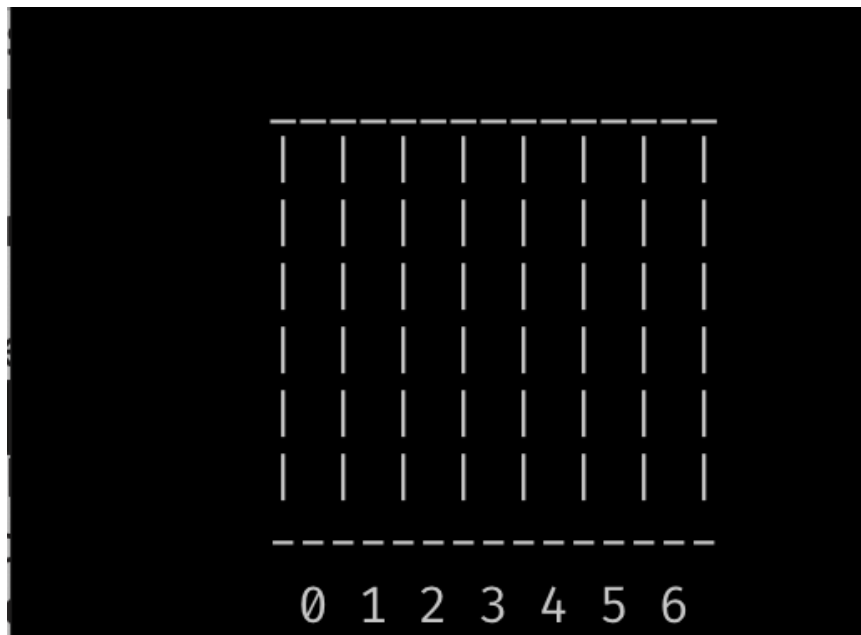
Affichage de la grille

- Crée une fonction `initialiserGrille()` qui utilisera la variable globale `grille` pour lui redonner sa valeur par défaut :

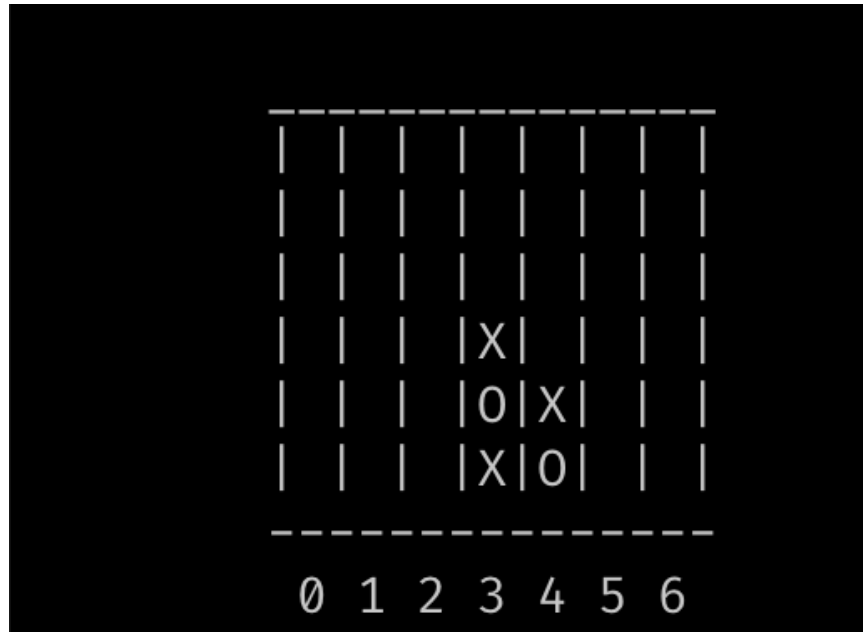
```
global grille
grille = [[0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0]]
```

- Crée une fonction `afficherGrille()`. Cette fonction doit simplement afficher toutes les cases de la grille `grille` (variable globale) pour que ce soit compréhensible pour les joueurs. Il faudra pour cela utiliser une double boucle pour parcourir chaque case de la liste. Pour chaque case, on doit afficher le symbole correspondant au pion du joueur placé. Dans notre cas, si c'est 1 il faudra afficher 'X'. Ensuite on peut ajouter des caractères pour bien séparer chaque case. **Voici un exemple d'affichage :**

Début de partie :



Après quelques tours :



Affichage de fin de partie

- Crée une fonction `afficherFinDePartie` qui prend en paramètre une variable `gagnant`. Cette fonction doit :
 - Afficher la grille.
 - Afficher un texte qui dit quel joueur a gagné. Dans le cas où c'est un match nul, `gagnant` aura comme valeur `0`. Ne pas oublier ce cas-là !

Tu peux tester tes fonctions d'affichage en les appelant dans le programme, pour être sûr que ce qui est affiché correspond bien à ce que tu voulais.

Étape 4 : Définir les fonctions pour lancer le jeu

- Crée une fonction `boucleDeJeu()` qui contiendra seulement le mot clef `pass`. Nous coderons cette fonction à l'étape suivante mais nous avons besoin de l'appeler dans la fonction de lancement du jeu.

```
def boucleDeJeu():  
    pass
```

- Crée une fonction `demandeRejouer()`. Elle devra :
 - Demander à l'utilisateur s'il veut rejouer tant que sa réponse n'est ni 'o' ni 'n'. Il devra taper 'o' s'il veut rejouer et 'n' s'il veut arrêter.
 - Retourner True si l'utilisateur veut rejouer et False sinon.
- Crée une fonction `jeu()` :
 - Tant que l'utilisateur veut rejouer, il faut :
 - * Initialiser la grille en utilisant la fonction adéquate.
 - * Appeler la fonction `boucleDeJeu()` et récupérer son résultat.
 - * Afficher le message de fin de partie en utilisant la fonction adéquate.
 - * Demander à l'utilisateur s'il veut rejouer en utilisant la fonction adéquate.
 - Quand on sort de la boucle, cela veut dire que l'utilisateur ne veut plus jouer. Dans ce cas, il faudra juste afficher Au revoir !.

Étape 5 : Déroulement d'une partie

Dans cette étape nous allons commencer la fonction principale `boucleDeJeu()`. Nous allons préparer la logique et nous coderons les fonctions intermédiaires au fur et à mesure des étapes suivantes.

Fonctions intermédiaires

Pour commencer, nous allons déclarer les fonctions utilisées dans `boucleDeJeu()` sans pour autant en écrire le code, avec le mot-clef `pass` :

- Crée une fonction `joueurGagne()` qui prend en paramètre `joueur`.
- Crée une fonction `verifierGrillePleine()`.
- Crée une fonction `jouerLeTour()` qui prend en paramètre `joueur`.

Boucle du jeu

Définissons maintenant la logique de la boucle de jeu :

- Modifie la fonction `boucleDeJeu()` en enlevant l'instruction `pass`
- Nous avons besoin d'une variable qui contiendra le numéro du joueur qui doit jouer. Cette variable sera toujours égale à 1 au début d'une partie (le joueur 1 commence).
- Tant que la grille n'est pas pleine (`verifierGrillePleine()`), il faut :
 - Afficher la grille.
 - Jouer le tour du joueur actuel (sans oublier de préciser quel joueur).
 - Si le joueur actuel gagne (`joueurGagne(...)`), il faut renvoyer la valeur du joueur gagnant.
 - Sinon, inverser le numéro du joueur (la variable devient 2 si on fait jouer 1 et vice-versa).
- Quand c'est fini, cela signifie qu'il y a match nul. Dans ce cas, il faudra renvoyer 0.

Étape 6 : Déroulement d'un tour

Nous allons maintenant coder le déroulement d'un tour. Voici ce qu'il se passe à chaque tour :

- On demande à un joueur de choisir la colonne où placer son jeton. On lui redemande tant qu'il choisit une colonne inexistante (<0 ou >6).
- On trouve quelle ligne est la première ligne libre (en partant du bas) de la colonne choisie.
- On place le jeton du joueur dans la grille, aux coordonnées ainsi trouvées.

Demander la colonne

Commençons par la fonction qui demande la colonne au joueur :

- Crée une fonction `demanderColonne()` qui prend en paramètre `joueur`. Cette fonction doit :

- Afficher un message demandant au joueur de choisir sa colonne (préciser à quel joueur on demande pour que ce soit plus clair).
- Récupérer son choix et le convertir en entier.
- Recommencer tant que le choix de colonne est incorrect (afficher un message précisant pourquoi on redemande au joueur un numéro de colonne).
- Retourner le choix de l'utilisateur.

Trouver la ligne où placer le jeton

Passons maintenant à la fonction qui trouve la ligne libre d'une colonne. Nous allons devoir coder une fonction qui parcourt chaque ligne de la grille pour vérifier si la colonne choisie est libre sur cette ligne. (Voir le fonctionnement de la grille à l'étape 1 pour savoir si une case est libre).

Il faudra garder la dernière ligne libre trouvée, car le parcours de la liste se fait de haut en bas et que nous devons placer les jetons le plus bas possible.

- Crée une fonction `ligneLibreDeLaColonne()` qui prend en paramètre la `colonne`. Cette fonction doit parcourir toutes les lignes de la `colonne` afin de trouver la colonne libre la plus basse. Elle devra renvoyer le numéro de la ligne libre et -1 si aucune ligne n'est libre pour cette `colonne`.

Placer le jeton

La dernière fonction intermédiaire pour un tour sera la fonction `placerJeton(joueur, coordonnees)`. Cette fonction placera simplement le numéro du `joueur` à un l'endroit défini par le tuple `coordonnees`.

Jouer le tour

Enfin, nous allons pouvoir créer la fonction qui fait jouer le tour au joueur :

- Modifie la fonction `jouerLeTour()` précédemment créée en supprimant le mot-clef `pass`. Voici la logique de la fonction :
 - Tant que la `ligne` vaut -1 (c'est à dire est incorrecte) :
 - * Demander la colonne au joueur (récupérer le résultat).
 - * Trouver la ligne libre pour cette colonne (récupérer le résultat).

- * Si la ligne vaut encore **-1** dire au joueur que la colonne qu'il a choisie n'est plus libre.
- Quand on a trouvé la ligne correspondant à la colonne choisie, placer le jeton du joueur aux coordonnées trouvées.

Nous avons fini de créer le déroulement d'un tour. Dans la prochaine étape, nous nous attaquerons à la fonction qui trouve le gagnant d'une partie.

Étape 7 : Y a-t-il un vainqueur ?

Désormais, on peut faire jouer un tour à un joueur. Il faut maintenant que l'on puisse déterminer à chaque fin de tour si le joueur qui vient de jouer a gagné ou s'il y a match nul.

Pour cela, nous utiliserons deux fonctions différentes : une première qui permet de vérifier si un joueur passé en paramètre a gagné et une seconde qui permet de vérifier si la grille est pleine.

Trouver un alignement

Nous allons commencer par déterminer si un joueur a gagné. Pour cela, nous devons créer 3 fonctions intermédiaires :

- **verifierHorizontalement(joueur)** : cette fonction vérifie si le joueur spécifié passé en paramètre a un **alignement horizontal** de 4 jetons. Pour cela, il faut parcourir toutes les lignes de la liste pour vérifier si le numéro du joueur apparaît 4 fois de suite dans la ligne.
- **verifierVerticalement(joueur)** : cette fonction vérifie si le joueur spécifié a un **alignement vertical** de 4 jetons. Il faudra parcourir chaque colonne de la liste pour vérifier si le numéro du joueur apparaît 4 fois de suite dans la colonne. **Conseil** : pas besoin de commencer la vérification au-delà de la 3^{ème} ligne, car ce n'est plus possible d'aligner 4 pions sur 3 lignes !
- **verifierDiagonale(joueur)** : cette fonction vérifie si le joueur spécifié a un **alignement en diagonale** de 4 jetons. Il faudra parcourir toute la liste en cherchant si le numéro du joueur apparaît 4 fois de suite dans une diagonale **ou dans l'autre**. En effet, n'oublie pas les deux sens possibles d'une diagonale ! Cette fonction sera un peu plus difficile que les autres, je te conseille de commencer par essayer de parcourir les diagonales de la liste en faisant des tests d'affichage.

Le joueur a-t-il gagné ?

Maintenant que nous avons codé nos 3 fonctions intermédiaires, nous pouvons modifier la fonction `joueurGagne(joueur)`. La fonction doit maintenant renvoyer `True` si le `joueur` a un alignement horizontal, vertical ou en diagonale. Sinon, elle retourne `False`.

Match nul ?

Enfin, il nous manque la fonction qui vérifie s'il y a **match nul**. Nous savons qu'il y a match nul si la grille est remplie et que personne n'a gagné. Alors, nous avons simplement besoin d'une fonction qui vérifie si la grille est pleine :

- Modifie la fonction `verifierGrillePleine()`. Elle doit renvoyer `True` si la grille est pleine et `False` sinon.

Étape 8 : Lancement du jeu !

Désormais, il ne nous reste plus qu'à lancer le jeu : appelle simplement la fonction `jeu()` dans le programme !

Quand tu vas tester le jeu, il est très fort probable qu'il y ait des bugs et que certaines choses ne fonctionnent pas : c'est normal pour un gros programme. Ne t'inquiètes pas et n'hésite pas à faire plein de tests pour chaque fonction pour trouver d'où vient le problème.

Intelligence Artificielle du Puissance 4

Théorie

Ressources

<https://www.youtube.com/watch?v=0lQxdR6IqCA&t=103s>

Un cours de fac plutôt bien résumé : MinMax

Code

Étape 1 : Tous les coups possibles

Crée une fonction qui doit retourner la liste de tous les coups possibles au prochain tour. Chaque élément de la liste sera un tuple correspondant à la ligne et la colonne où le jeton peut être placé : **(ligne, colonne)**.

Étape 2 : Tous les alignements

Crée une fonction qui retourne la liste de tous les alignements de 4 cases de la grille. La liste contiendra donc des listes de 4 éléments. Les alignements possibles sont :

- En ligne
- En colonne
- En diagonale vers la droite
- En diagonale vers la gauche

Étape 3 : Fonction d'évaluation

Crée une fonction qui devra attribuer un score à une grille pour un joueur donné. Cette fonction permettra à l'IA de savoir quel coup lui rapporte le plus de points. Elle sera utilisée plus tard dans la fonction chargée d'anticiper quel coup amènera au meilleur résultat. Voici quelques indications :

Le score varie en fonction des alignements de 4 encore possibles pour le joueur donné en paramètres. C'est ici que tu auras besoin de la fonction de l'étape 2. Sers-toi de la liste pour parcourir chaque alignement et vérifier si le joueur en paramètre peut encore aligner 4 pions dedans.

S'il peut, alors il faut compter combien de pions sont déjà présents dans l'alignement. Si c'est l'autre joueur qui peut, il faut aussi sauvegarder son score dans une autre variable. Attention, il ne faut pas ajouter 1,2 ou 3 au score car dans ce cas on ne peut pas savoir si 3 correspond à 3 pions alignés ou 3 pions dans 3 alignements différents. Pour contrer ce problème, voici la grille du score à ajouter en fonction du nombre de pions alignés :

Pions alignés	Score
0	+0
1	+1
2	+100
3	+10000
4	+1000000

Quand les scores des deux joueurs ont été calculés, la fonction devra renvoyer le score du joueur en paramètre moins le score de l'adversaire.

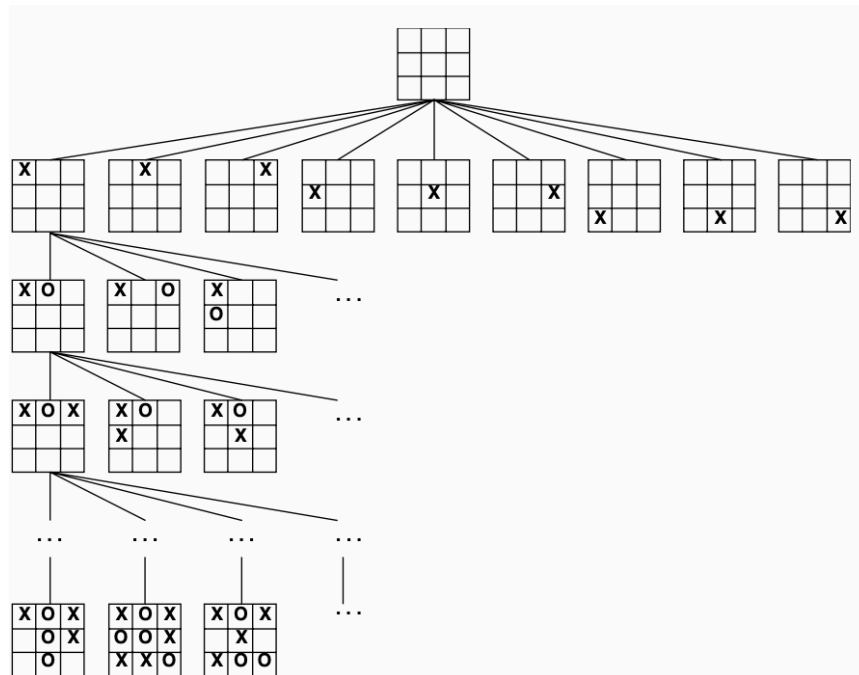
Un peu de cours

Recherche

Le jeu du Puissance 4 est un jeu dit **déterministe** et **complètement observable**. Cela signifie que l'agent sait dans quel état il est et dans quel état il sera (en fonction des actions qu'il fait). La solution de ce type de problème est une suite d'actions (par exemple jouer dans la colonne n°3).

Pour trouver la meilleure solution possible dans un état donné, l'agent doit connaître toutes les possibilités de jeu et leurs conséquences. Nous utilisons pour imaginer cela un **arbre de recherche**. Cet arbre de recherche permet de parcourir toutes les possibilités de jeu.

Exemple de l'arbre de recherche du morpion (ou tic-tac-toe) :



Pour pouvoir générer cet arbre de recherche pour notre IA, nous avons créé une fonction qui retourne tous les coups possibles, c'est-à-dire tous les ensembles (ligne, colonne) correspondants aux endroits où le prochain joueur peut placer son pion.

Algorithme Min-Max

L'algorithme Min-Max est un algorithme utilisé uniquement pour la programmation d'intelligences artificielles de jeux. Le principe est le suivant:

- Quand c'est au tour de l'agent de jouer (notre IA), il doit trouver la solution qui lui offrira le plus de chances de gagner. Pour ce faire, il va devoir **tester tous les coups** qu'il peut jouer dans l'**état actuel** de la grille.
- Pour chaque coup possible, il vérifie le résultat (a-t-il gagné ?) et va tester **tous les coups que l'adversaire pourra jouer ensuite**.
- L'agent continue de simuler la partie jusqu'à ce qu'il arrive dans un **état final**, c'est-à-dire un état où soit un des joueurs a gagné soit la grille est remplie.
- L'agent doit recommencer cette simulation de partie pour tous les coups possibles à chaque fois.

C'est ici que nous aurons besoin d'utiliser la récursion : l'agent devra tester toutes les parties possibles pour déterminer quel coup amène la partie vers sa victoire. Chaque étape ou étage dans l'arbre de recherche correspond à ce que l'on appelle **la profondeur**. Plus l'agent va en profondeur, plus le temps de recherche sera long car à chaque prondeur, cela ajoute encore plus de solutions à vérifier.

L'algorithme Min-Max est donc une manière de générer et explorer l'arbre de recherche associé à la partie de Puissance 4.

Le fonctionnement de l'algorithme est le suivant :

- Max et Min sont les deux joueurs.
- Max joue en premier, Min en second.
- Le joueur Max (qui correspond à l'agent) cherche à obtenir le score maximal avec son coup.
- Le joueur Min (qui correspond au joueur humain) cherche à obtenir le score minimal avec son coup.

Le Puissance 4 est un **jeu à somme nulle**. Nous pouvons traduire cela par le fait que :

- Quand le joueur 1 gagne, le joueur 2 a perdu.

- Quand le joueur 2 gagne, le joueur 1 a perdu.
- Quand il y a match nul, aucun des joueurs n'a gagné.

Nous avons déjà pu remarquer ça quand nous avons programmé les fonctions pour déterminer le vainqueur d'une partie : quand on trouve l'alignement d'un joueur, nous n'avons plus besoin de vérifier le reste de la grille.

Pour l'algorithme Min-Max, nous aurons donc trois scores possibles pour une grille :

- 1 si Max gagne.
- -1 si Min gagne.
- 0 si aucun des deux ne gagne.

Le pseudo-code de l'algorithme Min-Max se trouve à la page 12 du cours dans la partie **Ressources**.

Retour au code

Pour notre jeu, nous utiliserons deux fonctions : `min_max(joueur_max, is_tour_max, profondeur)` et `trouver_meilleur_coup(joueur_max, profondeur)`.

- La fonction `trouver_meilleur_coup(...)` doit :
 - Récupérer tous les coups possibles.
 - Initialiser `meilleur_score` à `-math.inf`.
 - Initialiser `meilleur_coup` à `None`.
 - Pour chaque coup possible :
 - * Placer le jeton dans la grille aux coordonnées du coup.
 - * Récupérer le score du coup grâce à `min_max`.
 - * Effacer le coup dans la grille.
 - * Garder en mémoire le meilleur score et le meilleur coup.
 - Il faudra retourner à la fin les coordonnées du meilleur coup trouvé.
- Pour la fonction `min_max(...)`, nous pouvons utiliser le pseudo-code en page 12 du cours dans **Ressources**.